HoGent

HoGent
BEDRIJF
EN
ORGANISATIE

2 TIN
# Databases

## Transaction management

## What do you learn in this chapter?

- **transactions**
  - function, relevance and properties
- **concurrency control**
  - deadlocks: detection and solutions
  - timestamps and serializability
  - application in MS SQL Server

## What do you learn in this chapter?

- **recovery**
  - causes and recovery in case of db failure
  - de transaction log file
  - checkpointing

## Introduction

- a **DBMS** supports
  - transactions
  - concurrency control services
  - recovery services
- goal
  - keep the db in a reliable and consistent state
    - in case of software, hardware failures
    - in case of simultaneous use by different users

## TRANSACTIONS

# Example: database xTreme

- What's the problem with this procedure?

```
/* Write a stored procedure for adding a
productClass.
Return the generated key (no identity)*/

create procedure demo @productclassname nvarchar(50)
as
declare @id int
select @id=max(productclassid) from productclass
set @id = @id + 1
insert into productclass
values(@id,@productclassname)
return @id
```

## What is a transaction ?

a **transaction** is
an action or a sequence of actions on a db
that have to be considered as **1 logical unit**

- action: reading or changing database content
- a transaction is executed by a user or an application
- logical unit of work
  - a complete application
  - part of an application
  - a single command
- application
  - 1 or more transactions with non db processing in between

# Example

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

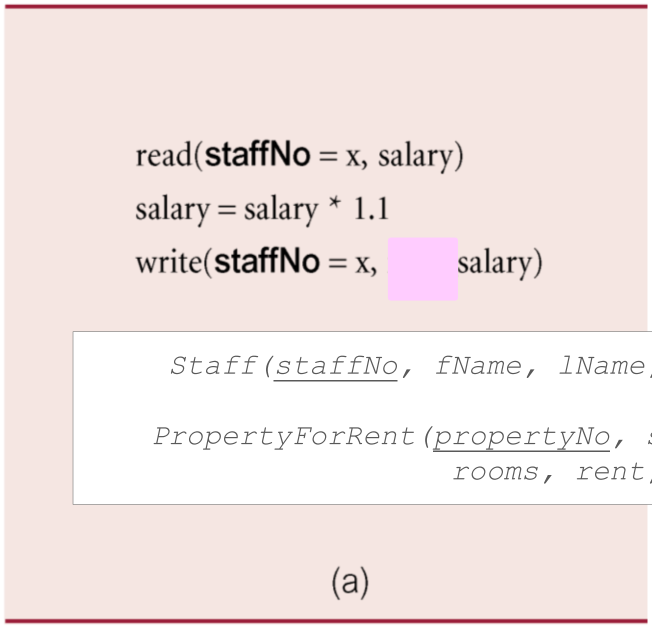
(a)

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)

```
Staff (staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent,
                                                ownerNo, staffNo, branchNo)
```

## Example



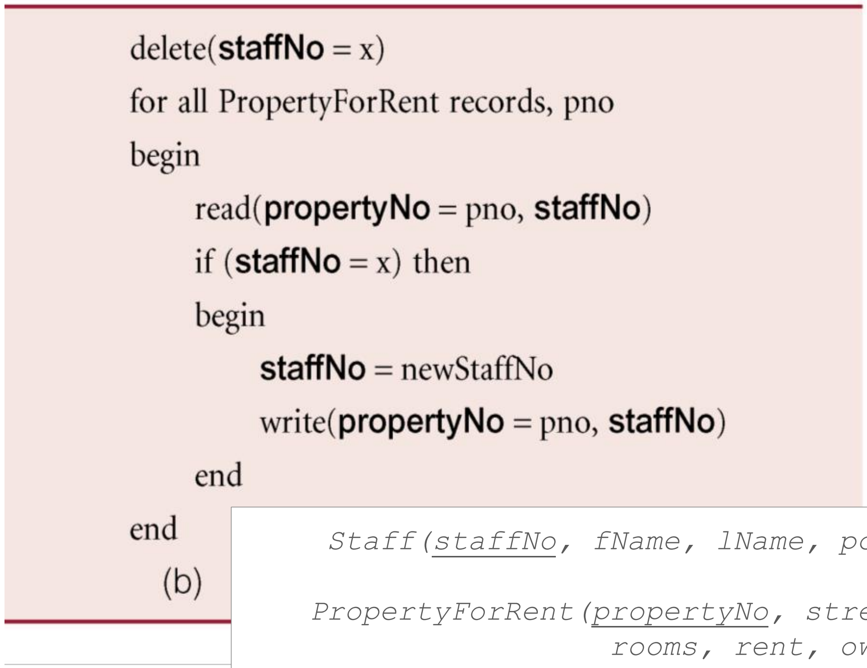
```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, salary)
```

(a)

*Staff(staffNo, fName, lName, position, sex, DOB, salary, branchNo)*
*PropertyForRent(propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)*

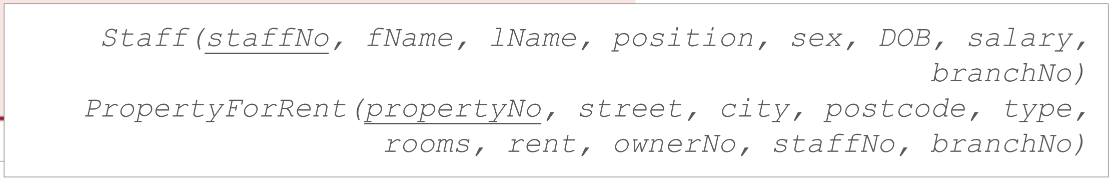
- transaction: updating the salary of an employee
- sequence of 'high level' action
  - notation: read(…) and write(…)  (= pseudo language)
    - read: read a data item from the db in a local variable
    - write: write the value of a local variable to the db
  - there are 2 db operations and 1 non db operations

## Example



```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

(b)



```
Staff(staffNo, fName, lName, position, sex, DOB, salary,
    branchNo)
PropertyForRent(propertyNo, street, city, postcode, type,
    rooms, rent, ownerNo, staffNo, branchNo)
```

- transaction: deleting an employee and assign his properties to another employee
- this transaction brings the database from a consistent state to another consistent state

**during** the transaction the db is possibly in an inconsistent state

- ex. a number of properties are already assigned to newStaffNo and the rest is still assigned to staffNo

## Result of a transaction

**committed / aborted**

- **committed**
  - the transaction was **succesful**
  - the transaction *commits* and the db has reached a consistent state
- **aborted**
  - the transaction was not **succesful**
  - the transaction *aborts* and the db has to be restored to the consistent state before the transaction started
  - **rollback or undo**

## Result of a transaction

- **a committed transaction can never be aborted**
  - if it appears the transaction was a mistake, another, compensating transaction, can be used.
- **an aborted transaction can be restarted**
  - after the **rollback** a **restart** of the transaction can be executed
  - successful execution will no lead to a committed transaction

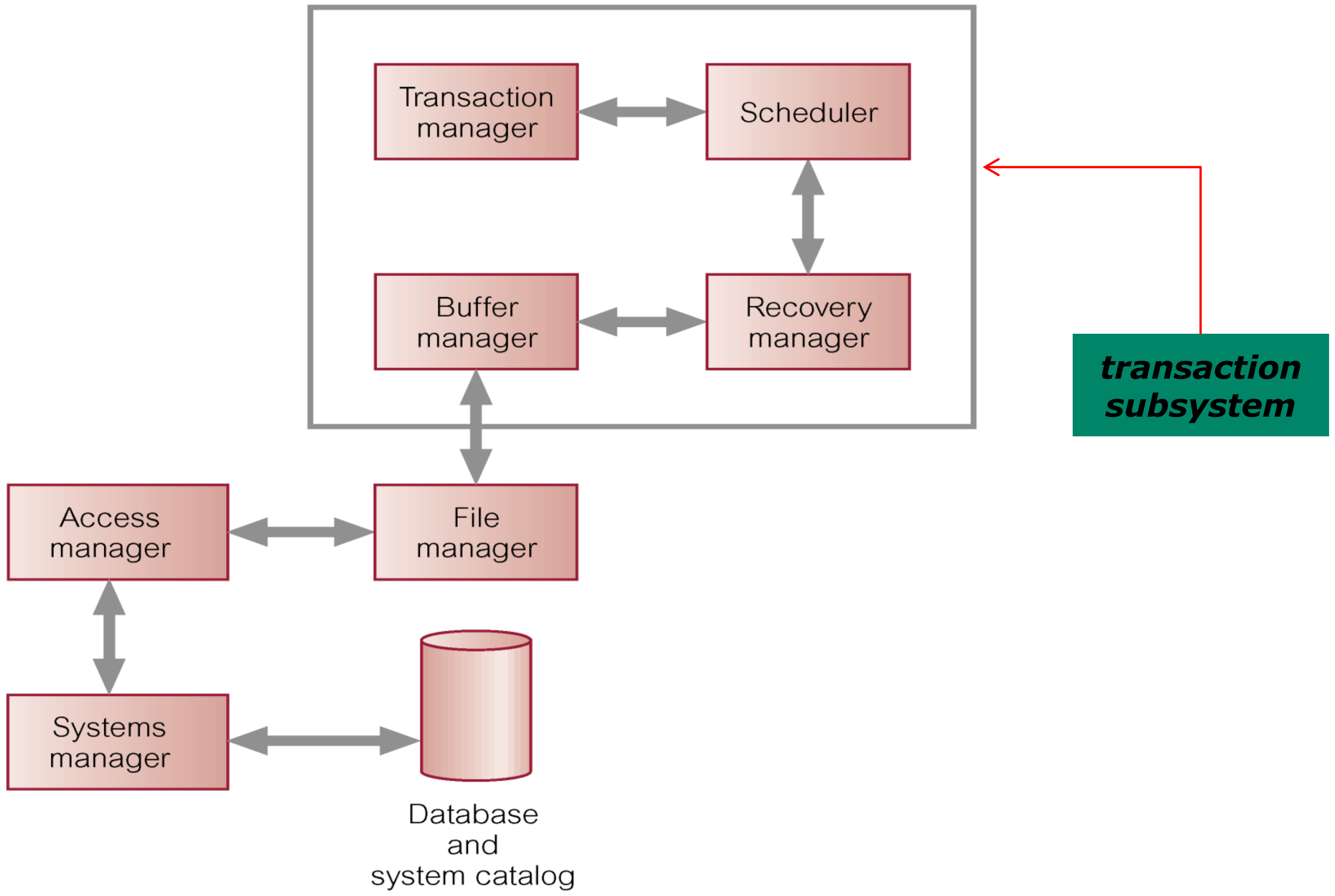
## Commands

- tell the DBMS which action are part of transation
- keywords
  - BEGIN TRANSACTION
    - sometimes implicitly with first db action in session
  - COMMIT
  - ROLLBACK
- without explicit command the complete program or session might be considered as a transaction

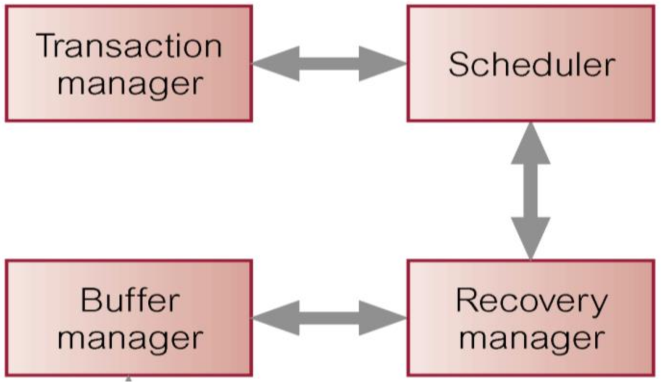
## ACID properties

- **Atomicity** – the commands in a transaction are considered as indivisible
  - all or nothing
  - responsibility of DBMS : recovery subsystem
- **Consistency** – a transaction transforms the db form one consistent state to another consistent state
  - responsibility of both DMBS and application
- **Isolation** - transactions are executed independently of one another
  - transactions should not have unwanted interferences with each other
  - partial effects of incomplete transactions should not be visible to other transactions
  - responsibility of DBMS: concurrency-control subsystem
- **Durability** - effects of a committed transaction are permanent and must not be lost because of later failure.
  - after a failure these changes can be recovered
  - responsibility of DBMS : recovery subsystem

## Database architecture



# Transaction subsystem

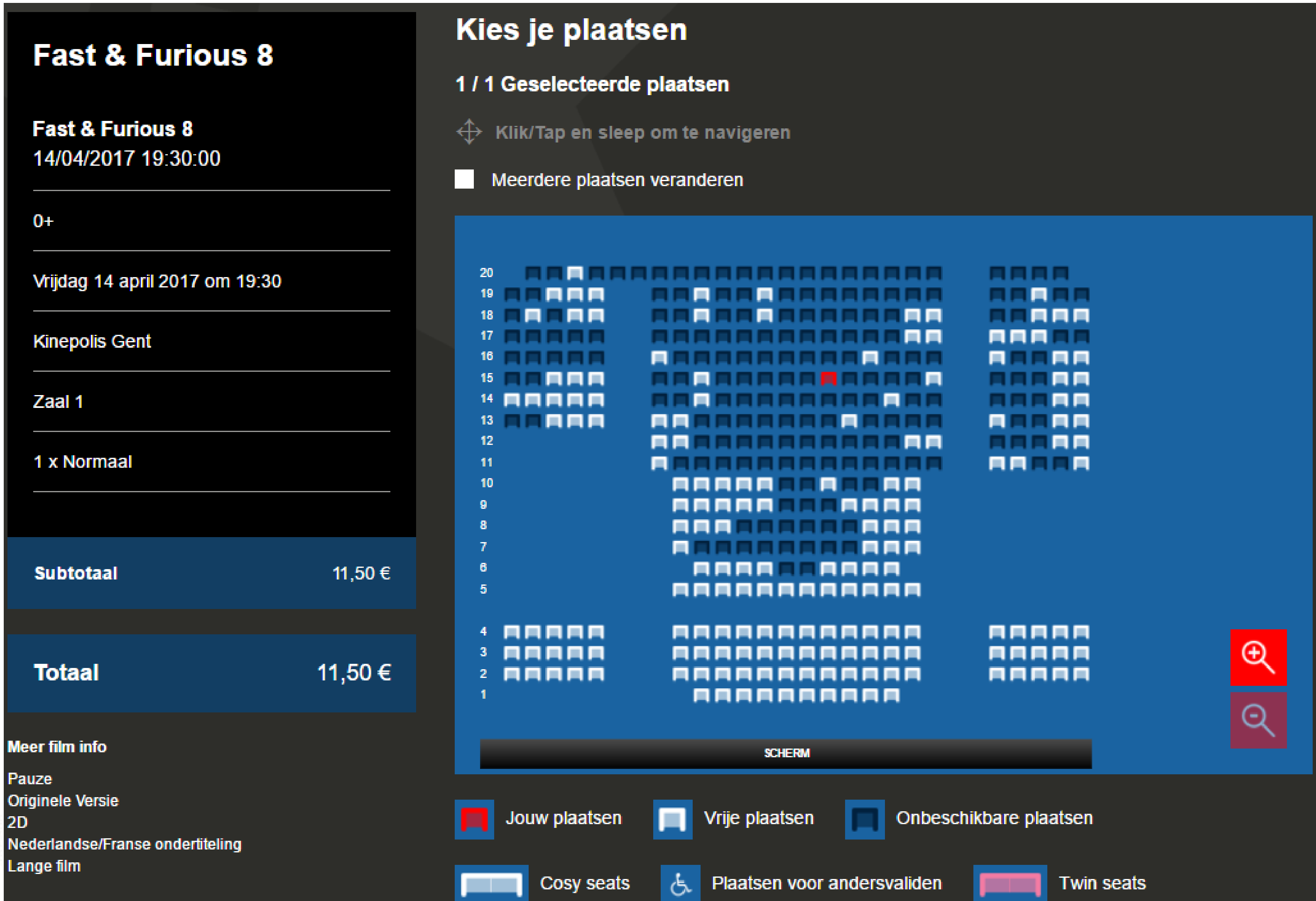


- **transaction manager**
  - coordinates transaction of applications
- **scheduler**
  - implements strategy for concurrency control
  - aka lock manager in case of locking based strategies (e.g. SQL Server)
  - goal: maximize concurrency without interference between transactions
- **recovery manager**
  - brings the DB back to consistent state in case of failure (~before transaction started)
- **buffer manager**
  - responsible for efficient transfer of data between main memory and disk storage

## CONCURRENCY

# Example: book a movie ticket

How to avoid that you have to share your seat?



**Fast & Furious 8**

**Fast & Furious 8**
14/04/2017 19:30:00

0+

Vrijdag 14 april 2017 om 19:30

Kinepolis Gent

Zaal 1

1 x Normaal

| | |
|---|---|
| **Subtotaal** | 11,50 € |
| **Totaal** | 11,50 € |

**Meer film info**

Pauze
Originele Versie
2D
Nederlandse/Franse ondertiteling
Lange film

**Kies je plaatsen**

**1 / 1 Geselecteerde plaatsen**

Klik/Tap en sleep om te navigeren

☐ Meerdere plaatsen veranderen

| | |
|---|---|
| 🟥 | Jouw plaatsen |
| ⬜ | Vrije plaatsen |
| 🟦 | Onbeschikbare plaatsen |
| ▭ | Cosy seats |
| ♿ | Plaatsen voor andersvaliden |
| ▭ | Twin seats |

HoGent

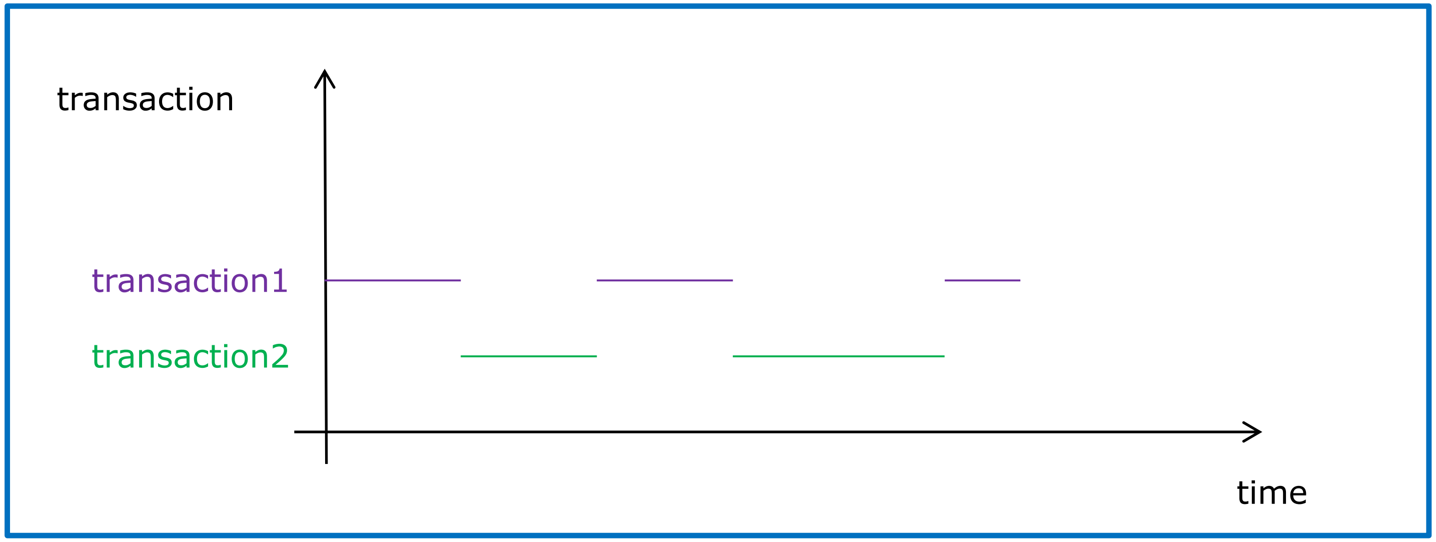
## What is concurrency control?

**concurrency control**
is the process of **managing simultaneous operations** on the database **without having them interfere** with one another.

- a DBMS allows that two or more transactions access shared data
  - if two or more transactions **only read data: no problems**
  - if at least 1 transaction changes **data: problems can occur**
- concurrency
  - interleaving of actions of different transactions

## Concurrency

- Example
  - actions of transaction 1 are executed, until an I/O operation occurs
  - the I/O operation is offered to the I/O subsystem
  - while waiting for the result actions from transaction 2 can be executed
  - transaction 2 reaches an I/O operation, which is again offered to the I/O subsystem
  - transaction 1 can continue, …
  - …



## Why concurrency control?

- Prevents interference when two or more users are accessing the database simul-taneously and at least one is updating data.

- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

## Problems with concurrency

- lost update
- uncommitted dependency
- inconsistent analysis

## Lost update

a **lost update problem** occurs when
**a successfully completed update is overridden by another user**

- Example with 2 transactions T1 and T2
  - balance on an account is 100 €
  - T1 withdraws 10 €
  - T2 deposits 100 €

  - in case of **serial** (non concurrent) execution the final balance is 190 €
    - T1 followed by T2, or
    - T2 followed by T1

## Lost update

| Time | $T_1$ | $T_2$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | begin_transaction | read($bal_x$) | 100 |
| $t_3$ | read($bal_x$) | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | $bal_x = bal_x - 10$ | write($bal_x$) | 200 |
| $t_5$ | write($bal_x$) | commit | 90 |
| $t_6$ | commit | | 90 |

- concurrent execution of T1 and T2
  - loss of T2's update
  - can be avoided by preventing T1 from reading $bal_x$ until after update.

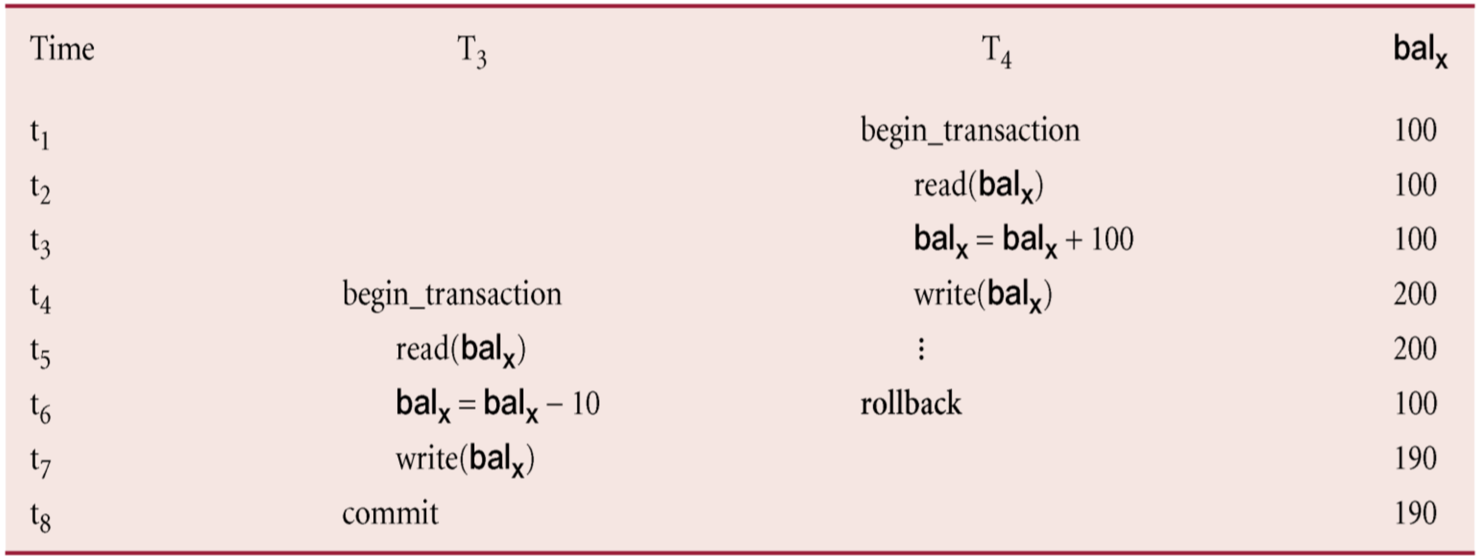
# Uncommitted dependency

the **uncommitted dependency (or dirty read) problem** occurs when one transaction can see intermediate results of another transaction before it has committed.

- Example with 2 transactions T3 and T4
  - balance on account is 100 €
  - T4 deposits 100 €
  - T4 aborts
  - T3 withdraws 10 €

  - in case of **serial** (non concurrent) execution the final balance is 90 €
    - T3 followed by T4, or
    - T4 followed by T3

## Uncommitted dependency



| Time | $T_3$ | $T_4$ | $bal_x$ |
|---|---|---|---|
| $t_1$ | | begin_transaction | 100 |
| $t_2$ | | read($bal_x$) | 100 |
| $t_3$ | | $bal_x = bal_x + 100$ | 100 |
| $t_4$ | begin_transaction | write($bal_x$) | 200 |
| $t_5$ | read($bal_x$) | ⋮ | 200 |
| $t_6$ | $bal_x = bal_x - 10$ | rollback | 100 |
| $t_7$ | write($bal_x$) | | 190 |
| $t_8$ | commit | | 190 |

- concurrent execution of T3 and T4
  - T3 supposes that T4 was succesful
  - but T4 performs a rollback…
- Problem avoided by
  - preventing T3 from reading $bal_x$ until after T4 commits or aborts.

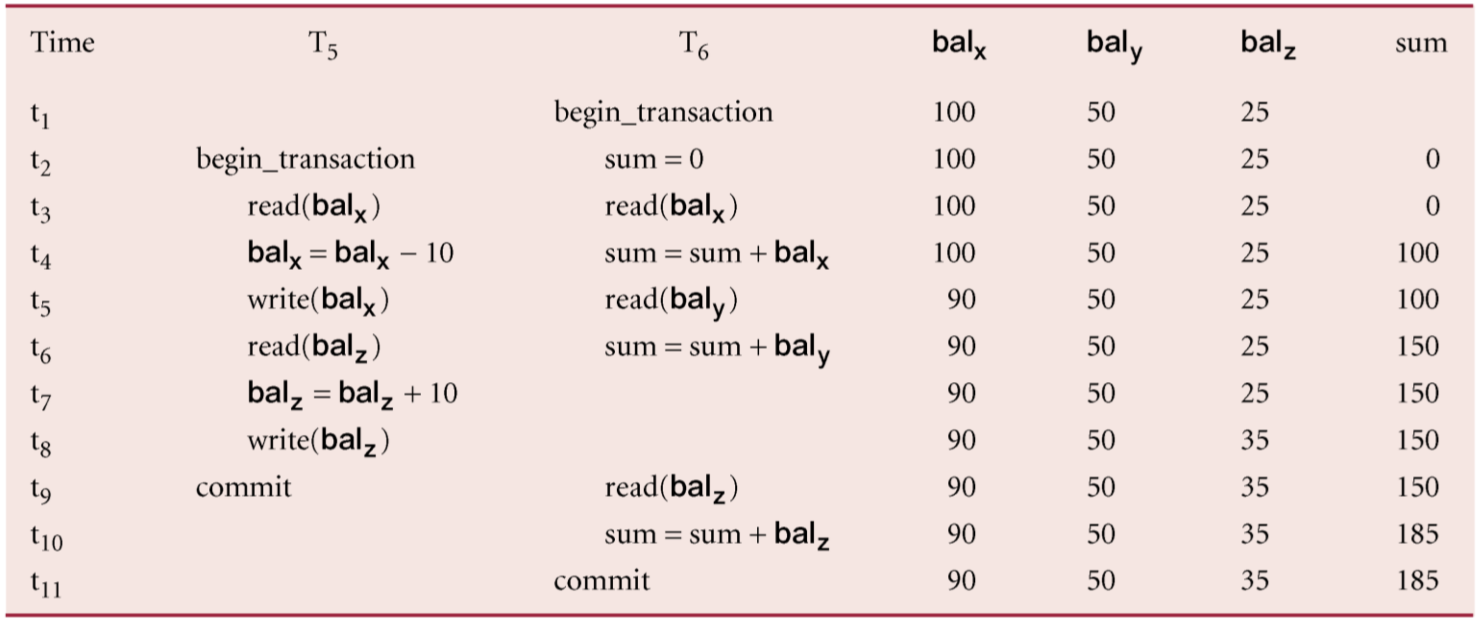
## Inconsistent analysis

the **inconsistent analysis probleem**
occurs when a transaction reads several values but a second transaction updates some of them during execution of the first.

- **Sometimes referred to as *unrepeatable read*.**
- Example with transactions T5 and T6
  - $T_6$ is totaling balances of account x (100 €), account y (50 €), and account z (25 €).
  - transaction T5 transfers 10 € from X to Z
  - in case of **serial** (non concurrent) execution the total 175 €
    - T5 followed by T6, or
    - T6 followed by T5

HoGent

## Inconsistent analysis



| Time | $T_5$ | $T_6$ | $bal_x$ | $bal_y$ | $bal_z$ | sum |
|---|---|---|---|---|---|---|
| $t_1$ | | begin_transaction | 100 | 50 | 25 | |
| $t_2$ | begin_transaction | sum = 0 | 100 | 50 | 25 | 0 |
| $t_3$ | read($bal_x$) | read($bal_x$) | 100 | 50 | 25 | 0 |
| $t_4$ | $bal_x = bal_x - 10$ | $sum = sum + bal_x$ | 100 | 50 | 25 | 100 |
| $t_5$ | write($bal_x$) | read($bal_y$) | 90 | 50 | 25 | 100 |
| $t_6$ | read($bal_z$) | $sum = sum + bal_y$ | 90 | 50 | 25 | 150 |
| $t_7$ | $bal_z = bal_z + 10$ | | 90 | 50 | 25 | 150 |
| $t_8$ | write($bal_z$) | | 90 | 50 | 35 | 150 |
| $t_9$ | commit | read($bal_z$) | 90 | 50 | 35 | 150 |
| $t_{10}$ | | $sum = sum + bal_z$ | 90 | 50 | 35 | 185 |
| $t_{11}$ | | commit | 90 | 50 | 35 | 185 |

- concurrent execution of T5 and T6
  - the total is calculated incorrectly as 185 €
- this can be avoided
  - by preventing $T_6$ from reading $bal_x$ and $bal_z$ until after $T_5$ completed updates.

## Inconsistent analysis

- related problems
  - **non-repeatable (or fuzzy) read**
    - reading twice the same item returns 2 different values: a transaction reads a data item a second time and gets another value
  - **phantom read**
    - a transaction reads a number of records that correspond to a certain condition (WHERE-clause)
    - the transaction repeats this later on but now gets extra records that meanwhile have been added by another transaction.

## Serializability & recoverability

how can we tackle these problems ?

## Serializability & recoverability

- concurrency control protocol
  - objective: schedule transactions in such a way as to avoid any interference
  - result: previous problems don't occurs
- trivial solution
  - run transactions serially
    - at any time only one transaction is running
    - major disadvantage: performance
      - limits degree of concurrency or parallelism in system
- serializability
  - identifies those executions of transactions guaranteed to ensure consistency.
    - executions are equivalent to serial executions

## Schedule

a **schedule** is
a **sequence of actions from a number of concurrent transactions** that respects the orders of the actions in the individual transactions

- transaction
  - sequence of operations, i.e. read/write actions on the db
- schedule
  - sequence of operations from a set of transactions
  - for each transaction in a schedule the order of operations in the schedule, is the same as the order of operations in the transaction itself, but it might be interleaved by operations from other transactions

## Recoverability

- serializability
  - the database makes schedules that can be executed without interference problems between transactions
  - the schedules guarantees consistency and isolation of transactions
  - assuming no transaction fails
- recoverability
  - if transaction fails, atomicity requires effects of transaction to be undone
  - durability states that once transaction commits, its changes cannot be undone (without running another, compensating, transaction).

==> **A C I D**

## Locking

locking is a method used to manage concurrent access to data.

- Transaction uses locks to deny access to other transactions and so prevent incorrect updates.
- Most widely used approach to ensure serializability.

## Locking

- **basic locking rules**
  - a transaction can read or write a data item if and only if it has acquired a lock for that data item, and it has not yet released that lock.
  - if a transaction acquires a lock on an item, it has to release that lock later on.

## Sort of locks

- **shared lock (S-lock)**
  - If transaction has shared lock on item, can read but not update item.
  - Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.

- **exclusive lock (X-lock)**
  - If transaction has exclusive lock on item, can both read and update item.
  - Exclusive lock gives transaction exclusive access to that item.

*remark: data item = field in record, complete record (default in SQL server), complete table or complete db (=granularity)*

# Locking

- **practical**:
  - a transaction that wants to access a data items **claims a lock** on that item
    - shared lock for reading, exclusive lock for reading/writing
    - if there is no existing lock on that item --> claim is granted
    - if there is already a lock --> dbms check is existing lock is **compatible**
      - claim of shared lock on item with shared lock: granted
      - claim of exclusive lock op item with lock: wait
        - » the transaction has to wait until the lock on the data item is released
  - the transaction **releases a lock**
    - explicitly, or
    - implicitly if the transaction ends (via abort or commit)

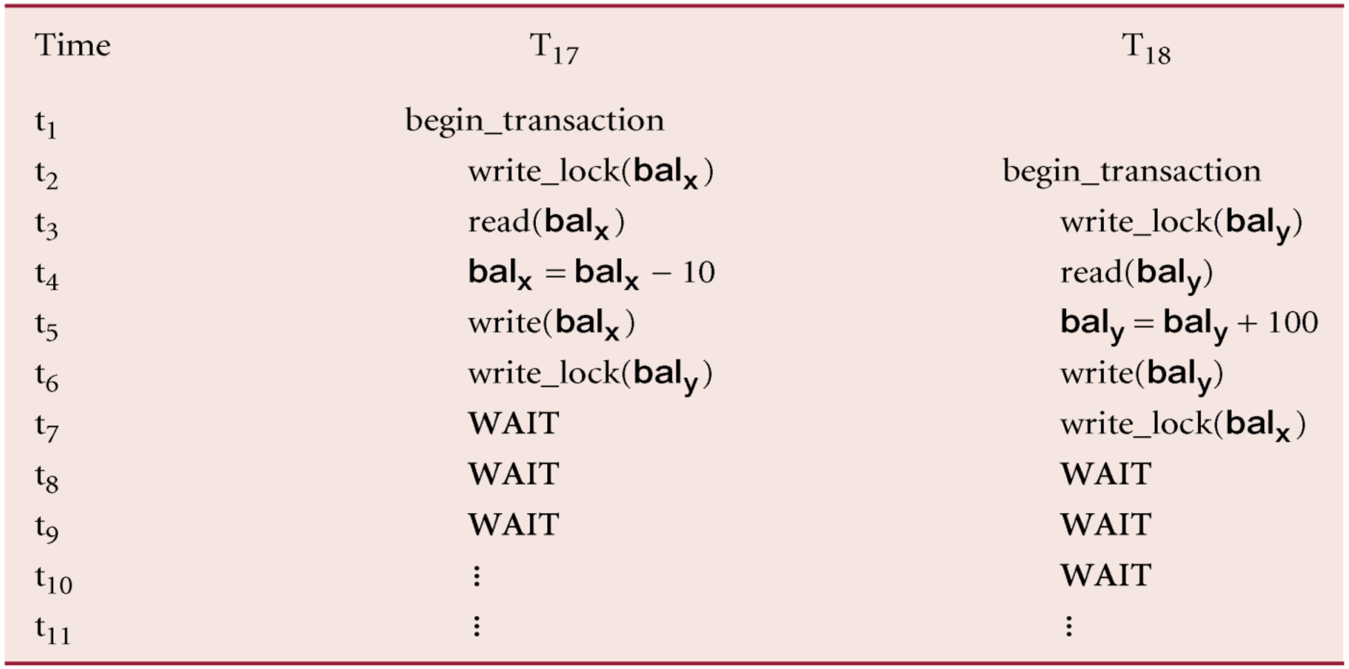
*claimed lock on data item*

| *existing lock on data item* | **S** | **X** |
|---|---|---|
| **S** | grant | wait |
| **X** | wait | wait |

*compatibilty matrix*

# Deadlock

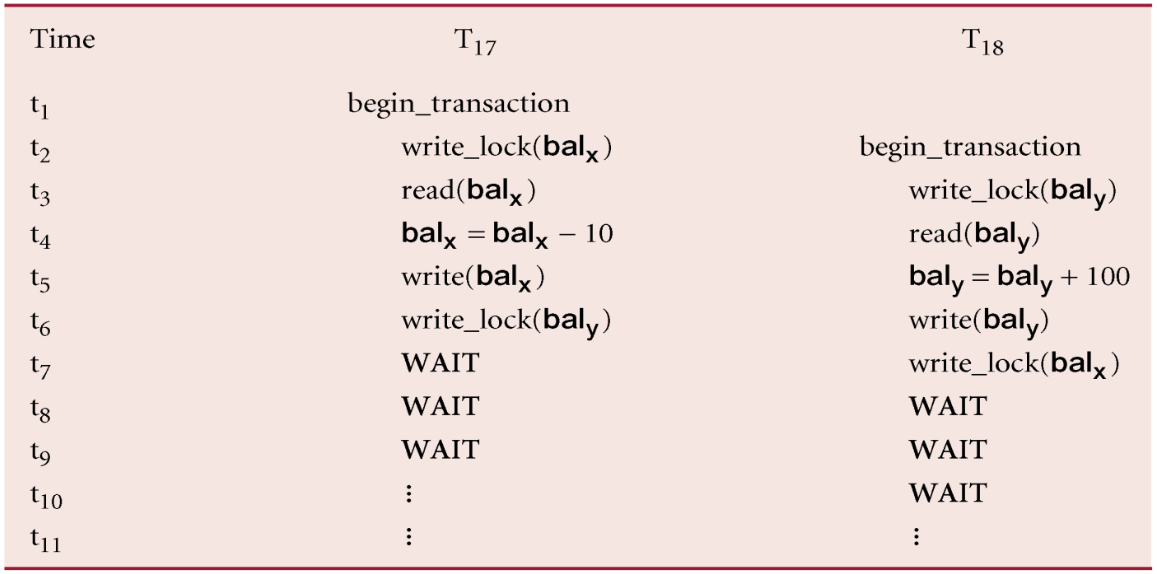
An **impasse** that may result when two (or more) transactions are each waiting for locks held by the other to be released.



| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock(**$\mathbf{bal_x}$**) | begin_transaction |
| $t_3$ | read(**$\mathbf{bal_x}$**) | write_lock(**$\mathbf{bal_y}$**) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read(**$\mathbf{bal_y}$**) |
| $t_5$ | write(**$\mathbf{bal_x}$**) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock(**$\mathbf{bal_y}$**) | write(**$\mathbf{bal_y}$**) |
| $t_7$ | WAIT | write_lock(**$\mathbf{bal_x}$**) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |

# Deadlock

- solving a deadlock
  - abort and restart of 1 or more transactions
  - Example:



| Time | $T_{17}$ | $T_{18}$ |
|---|---|---|
| $t_1$ | begin_transaction | |
| $t_2$ | write_lock($\mathbf{bal_x}$) | begin_transaction |
| $t_3$ | read($\mathbf{bal_x}$) | write_lock($\mathbf{bal_y}$) |
| $t_4$ | $\mathbf{bal_x} = \mathbf{bal_x} - 10$ | read($\mathbf{bal_y}$) |
| $t_5$ | write($\mathbf{bal_x}$) | $\mathbf{bal_y} = \mathbf{bal_y} + 100$ |
| $t_6$ | write_lock($\mathbf{bal_y}$) | write($\mathbf{bal_y}$) |
| $t_7$ | WAIT | write_lock($\mathbf{bal_x}$) |
| $t_8$ | WAIT | WAIT |
| $t_9$ | WAIT | WAIT |
| $t_{10}$ | ⋮ | WAIT |
| $t_{11}$ | ⋮ | ⋮ |

solution deadlock situation:
- abort T18
- the locks held by T18 are released
- T17 continues
- T18 is restarted automatically

## Deadlock

- approaches to handle deadlocks
  - timeouts
  - deadlock prevention
  - deadlock detection and recovery

## Timeouts

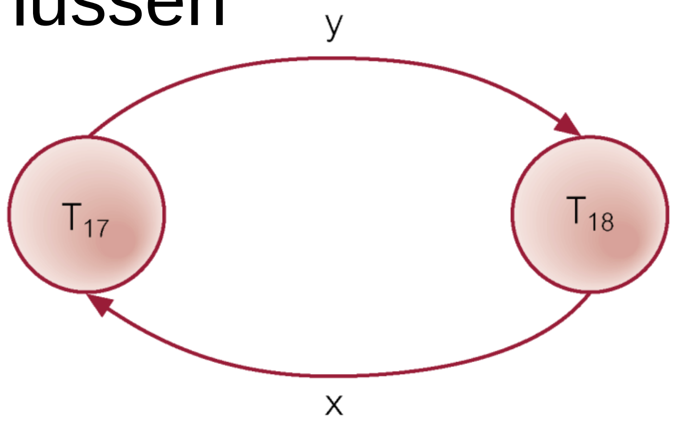
- timeouts
  - een transaction die een lock aanvraagt wacht voor een bepaalde vooraf gedefinieerde tijd op die lock
  - indien lock niet toegekend tijdens dit interval
    - assumptie dat er een deadlock is
      - hoewel dit niet noodzakelijk zo is…
    - abort and restart van de transaction

# Deadlock

- deadlock prevention
  - gebruik makend van transaction time-stamps
    - speciale time-stamp voor deadlock detection
  - T wacht op locks die U vastheeft
    - **wait-die algoritme**
      - als T ouder is dan U dan zal T wachten
      - in ander geval 'sterft' T
        - » abort/restart met dezelfde timestamp
    - **wound-wait algoritme**
      - als T ouder is dan U dan zal het U 'verwonden'
        - » meestal betekent dit een abort/restart van U
      - in andere geval zal T wachten

# Deadlock

- deadlock detection
  - gebruik makend van een wait-for graph met transaction afhankelijkheden
  - wanneer de wait-for graph een lus bevat is er een deadlock
  - op regelmatige tijdstippen wordt deze graph getest op lussen



## Deadlock

- recovery van deadlock detection
  - 1 of meerdere transactions worden ge-abort
- problemen
  - victim selection
    - abort die transaction waarvoor de abort een 'minimale kost' met zich meebrengt
    - enkele parameters die kunnen gebruikt worden:
      - tijd dat de transaction al aan het runnen was
      - aantal data items dat reeds werd gewijzigd door de transaction
      - aantal data items die nog moeten worden gewijzigd door de transaction
        - » deze parameter is echter niet altijd gekend

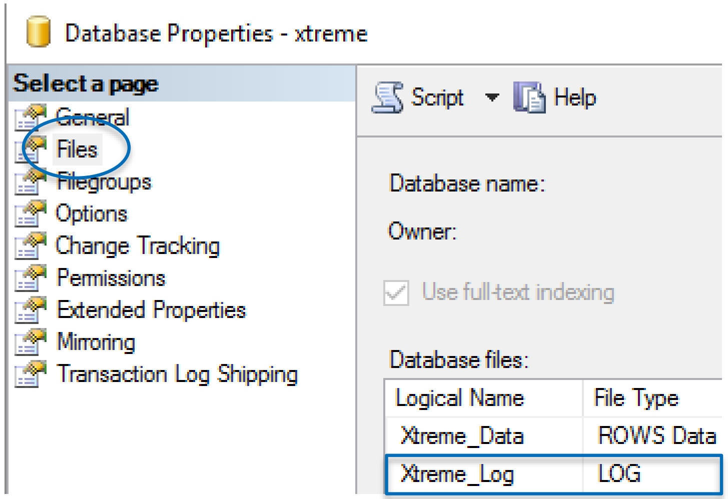
# Deadlock

- problemen *vervolg*
  - hoe ver moet een transaction een rollback doen
    - dit is niet noodzakelijk de volledige transaction
  - voorkomen van starvation
    - komt voor wanneer steeds dezelfde transaction als victim wordt geselecteerd
      - bijhouden van een teller

## TRANSACTIONS IN SQL-SERVER

# db transactions

- **impliciete** transactions
  - insert, update, delete
  - each transact sql command
- **explicit** transactions
  - developer determines when transaction starts and stop or how steps are rolled back to handle faulty situations
    - BEGIN TRANSACTION
    - COMMIT TRANSACTION
    - ROLLBACK TRANSACTION
- all info about transactions is written in the **transaction log** (see below)



Database name:

Owner:

Use full-text indexing

Database files:

| Logical Name | File Type |
|---|---|
| Xtreme_Data | ROWS Data |
| Xtreme_Log | LOG |

HoGent

## SP and transactions

```
CREATE PROCEDURE usp_Customer_Insert
          @customerid varchar(5),
          @companyname varchar(25)
          @orderid int OUTPUT
AS
  BEGIN TRANSACTION
    INSERT INTO customers(customerid, companyname)
    VALUES(@customerid, @companyname)
    if @@error <> 0 BEGIN
       ROLLBACK TRANSACTION
       RETURN -1
    END
    INSERT INTO orders(customerid)
    VALUES(@customerid)
    if @@error <> 0 BEGIN
       ROLLBACK TRANSACTION
       RETURN -1
    END
  COMMIT TRANSACTION
  SELECT @orderid=@@IDENTITY
```

***toevoegen van een klant and bestelling***

# Triggers and transactions

- a trigger is part of the same transaction as the triggering instruction
- inside the trigger this transaction can be ROLLBACKed
- ex. a player who is team coach can never be deleted (suppose there are no foreign key constraints)

```
CREATE TRIGGER delplayer ON PLAYERS
FOR delete
AS
IF (SELECT COUNT(*)
    FROM deleted JOIN teams
    ON teams.playerno = deleted.playerno) > 0
BEGIN
  ROLLBACK TRANSACTION
  RAISERROR ('The record does not exist.', 14,1)
END
```

## DEMO

- two "simultaneous" transactions with locking and victim selection

## Isolation levels

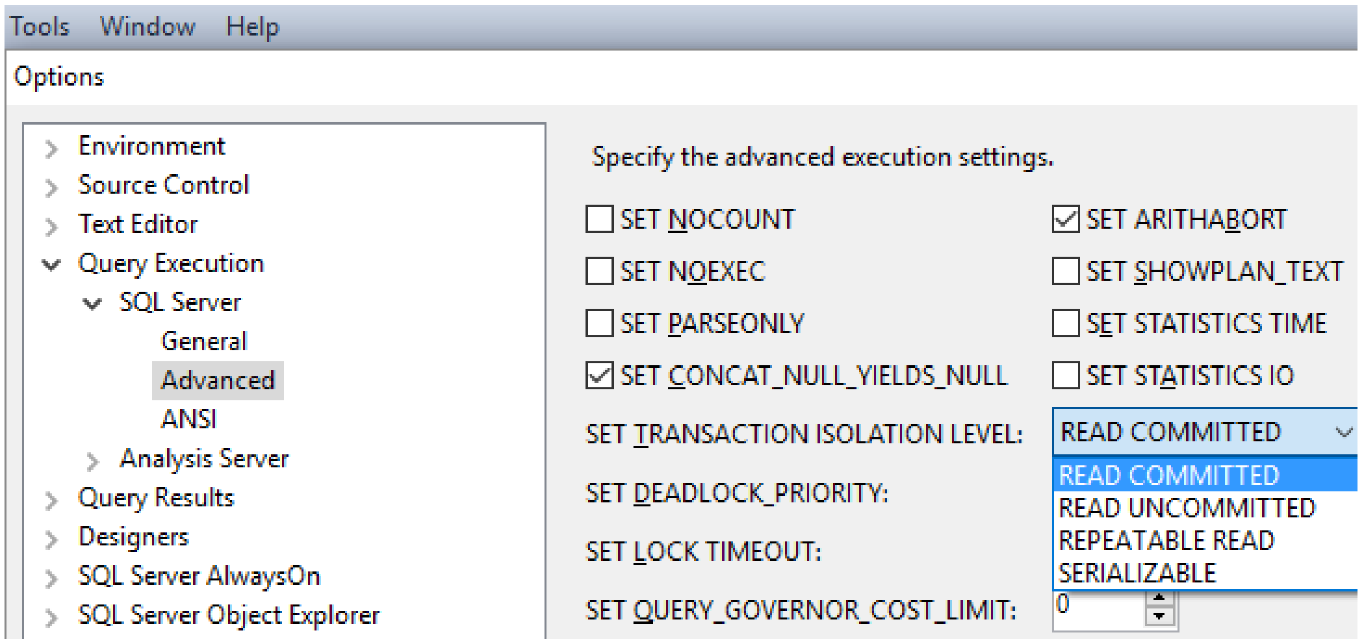
- Determine the behaviour of concurrent users who read or write data
- Reader: statement that reads data using a shared lock
- Writer: statement that writes data, using an exclusive lock
- Writers can't be influenced in SQL Server with respect to the locks they claim and the duration of these locks. They always claim an exclusive lock.
- Readers can be influenced explicitly
  - using *isolation levels*
  - this might have an implicit influence on the behaviour of writers
- Isolation level = setting at session or query level

# 4 isolation levels in SQL Server

Based on a pessimistic concurrency control (locking)

1. READ UNCOMMITTED
2. READ COMMITTED (default)
3. REPEATABLE READ
4. SERIALIZABLE

- locks take longer
- more consistentency
- less concurrency



Specify the advanced execution settings.

| | |
|---|---|
| ☐ SET NOCOUNT | ☑ SET ARITHABORT |
| ☐ SET NOEXEC | ☐ SET SHOWPLAN_TEXT |
| ☐ SET PARSEONLY | ☐ SET STATISTICS TIME |
| ☑ SET CONCAT_NULL_YIELDS_NULL | ☐ SET STATISTICS IO |
| SET TRANSACTION ISOLATION LEVEL: | READ COMMITTED |
| SET DEADLOCK_PRIORITY: | |
| SET LOCK TIMEOUT: | |
| SET QUERY_GOVERNOR_COST_LIMIT: | 0 |

READ COMMITTED
READ UNCOMMITTED
REPEATABLE READ
SERIALIZABLE

HoGent

## 1. Read uncommitted

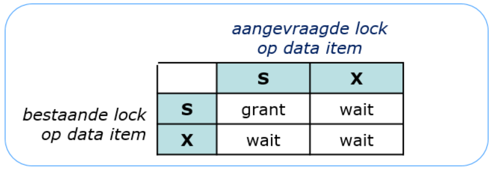
- lowest isolation level
- reader doesn't ask for *shared lock*
- reader never in conflict with writer (that holds exclusive lock)
- reader reads uncommitted data (= dirty read)

# 2. Read committed

- default *isolaton level,* see demo deadlocks
- lowest level that prevents dirty reads
- reader reads only committed data
- reader claims *shared lock*
- if at that moment a writer holds an *exclusive lock*, reader has to wait for *shared lock*
- reader keeps shared lock unit data is obtained (end of SELECT), not until end of transaction
  – reading again of data in same transaction can give different result
  = *non-repeatable reads* or *inconsistent analysis*
  – acceptable for many, but not all, applications

## 3. Repeatable read

- reader claims shared lock and holds it until end of  transaction
- other transaction can't get exclusive lock until end of transaction of reader



| | | *aangevraagde lock op data item* | |
|---|---|---|---|
| | | **S** | **X** |
| *bestaande lock op data item* | **S** | grant | wait |
| | **X** | wait | wait |

- *repeatable read* = *consistent analysis*
- also avoids *lost update* (possible in 1 & 2) by claiming shared lock at begin transaction (using SELECT because only readers can be influenced, not writers)

## 4. Serializable

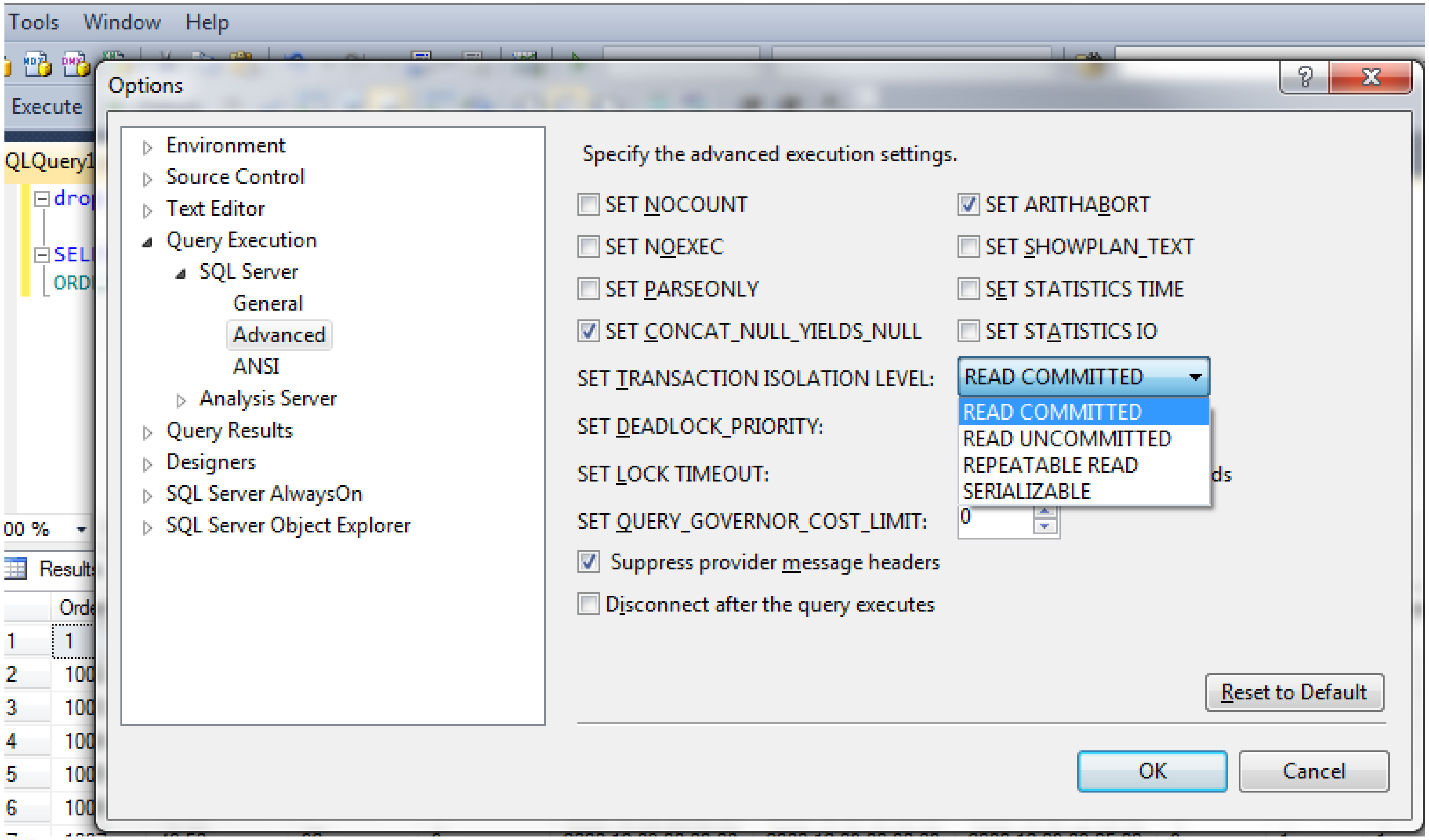
- Repeatable read only locks rows found with first SELECT
- Same SELECT in same transaction can give new row (added by other transactions) = *phantoms*
- *Serializable* avoids phantoms
- Lock all keys that correspond to WHERE-clause

# Isolation levels: session level

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED

or

- Via SSMS:



Specify the advanced execution settings.

| | |
|---|---|
| ☐ SET NOCOUNT | ☑ SET ARITHABORT |
| ☐ SET NOEXEC | ☐ SET SHOWPLAN_TEXT |
| ☐ SET PARSEONLY | ☐ SET STATISTICS TIME |
| ☑ SET CONCAT_NULL_YIELDS_NULL | ☐ SET STATISTICS IO |
| SET TRANSACTION ISOLATION LEVEL: | READ COMMITTED |
| SET DEADLOCK_PRIORITY: | |
| SET LOCK TIMEOUT: | |
| SET QUERY_GOVERNOR_COST_LIMIT: | 0 |

Dropdown options: READ COMMITTED, READ UNCOMMITTED, REPEATABLE READ, SERIALIZABLE

☑ Suppress provider message headers

☐ Disconnect after the query executes

HoGent

## Isolation levels: query level

- override isolation level with "table hint":

```
SELECT * FROM ORDERS WITH
(READUNCOMMITTED);
```

```
of
```

```
SELECT * FROM ORDERS WITH (NOLOCK);
```

- this avoids that long running queries on a production system lock updates in other transactions in case of READ COMMITTED and higher

# Example xTreme: solution using transactions

```
alter procedure example @productclassname nvarchar(50)
as
declare @id int
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
BEGIN TRANSACTION
-- Ensures shared lock on complete table.
-- Level "serializable" blocks adding new records during
-- transaction. Repeatable isn't sufficient:
-- only locks data already read
select @id=max(productclassid) from productclass
set @id = @id + 1
insert into productclass values(@id,@productclassname)
COMMIT
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
return @id
```

## RECOVERY

# Stroomloos Netlog beleeft helse 1 april

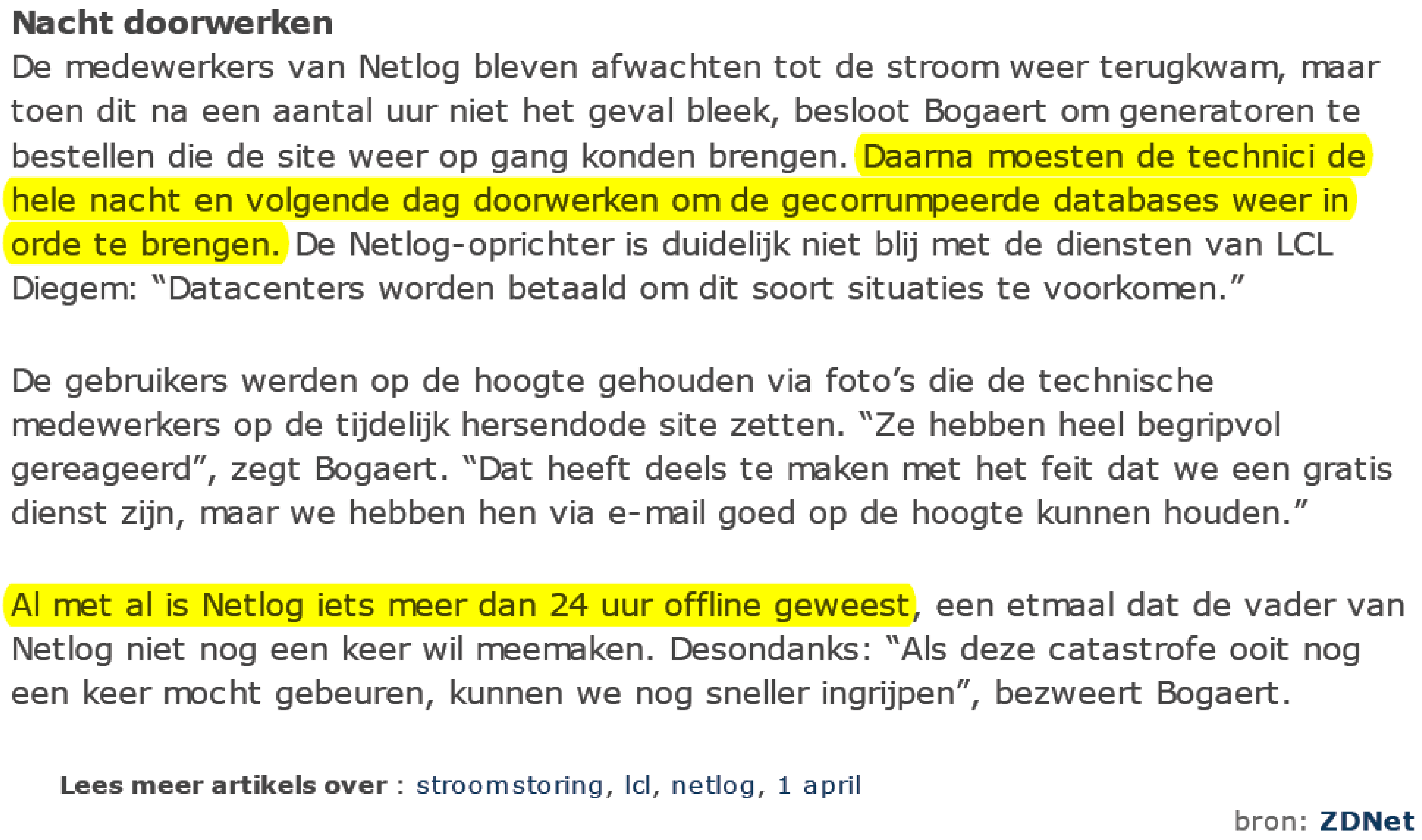
## "Dertien uur offline is heel lang voor een gebruiker"

11 april 2008 | Rowald Pruyn

Miljoenen gebruikers van Netlog dachten aan een 1 aprilgrap toen ze bij het ontwaken niet meer konden inloggen. De netwerksite zelf kon niet lachen om de stroomstoring die alle aangesloten landen een etmaal platlegde.

Datacenter **LCL Colocation** in Diegem, een van de grootste stroomvoorzieners in België, had last van een historisch lange stroomstoring die meer dan veertig procent van de Belgische websites urenlang platlegde. Netlog, dat ruim dertig miljoen gebruikers heeft in heel Europa, had plotseling te stellen met een sociaal netwerk dat van de een op de andere tel geheel offline ging.

“We hadden de pech dat we op 31 maart offline gingen, dus toen mensen op 1 april wilden inloggen, leek het alsof Netlog van de aardbodem was verdwenen”, vertelt Netlog-oprichter Lorenz Bogaert over de noodlottige nacht. Al snel stroomden de ongeruste telefoontjes binnen en vulden de **blogs** zich met wilde geruchten over het lot van de site. De verdwijning leidde zelfs tot huilende gebruikers die zonder thuishonk zaten. “Mensen probeerden op alle mogelijke manieren in contact te komen. We kregen telefoontjes uit Italië, Zwitserland; eigenlijk uit heel Europa”, vertelt Bogaert. “Dertien uur offline is heel lang voor een gebruiker.”



**Nacht doorwerken**

De medewerkers van Netlog bleven afwachten tot de stroom weer terugkwam, maar toen dit na een aantal uur niet het geval bleek, besloot Bogaert om generatoren te bestellen die de site weer op gang konden brengen. Daarna moesten de technici de hele nacht en volgende dag doorwerken om de gecorrumpeerde databases weer in orde te brengen. De Netlog-oprichter is duidelijk niet blij met de diensten van LCL Diegem: "Datacenters worden betaald om dit soort situaties te voorkomen."

De gebruikers werden op de hoogte gehouden via foto's die de technische medewerkers op de tijdelijk hersendode site zetten. "Ze hebben heel begripvol gereageerd", zegt Bogaert. "Dat heeft deels te maken met het feit dat we een gratis dienst zijn, maar we hebben hen via e-mail goed op de hoogte kunnen houden."

Al met al is Netlog iets meer dan 24 uur offline geweest, een etmaal dat de vader van Netlog niet nog een keer wil meemaken. Desondanks: "Als deze catastrofe ooit nog een keer mocht gebeuren, kunnen we nog sneller ingrijpen", bezweert Bogaert.

**Lees meer artikels over** : stroomstoring, lcl, netlog, 1 april

bron: **ZDNet**

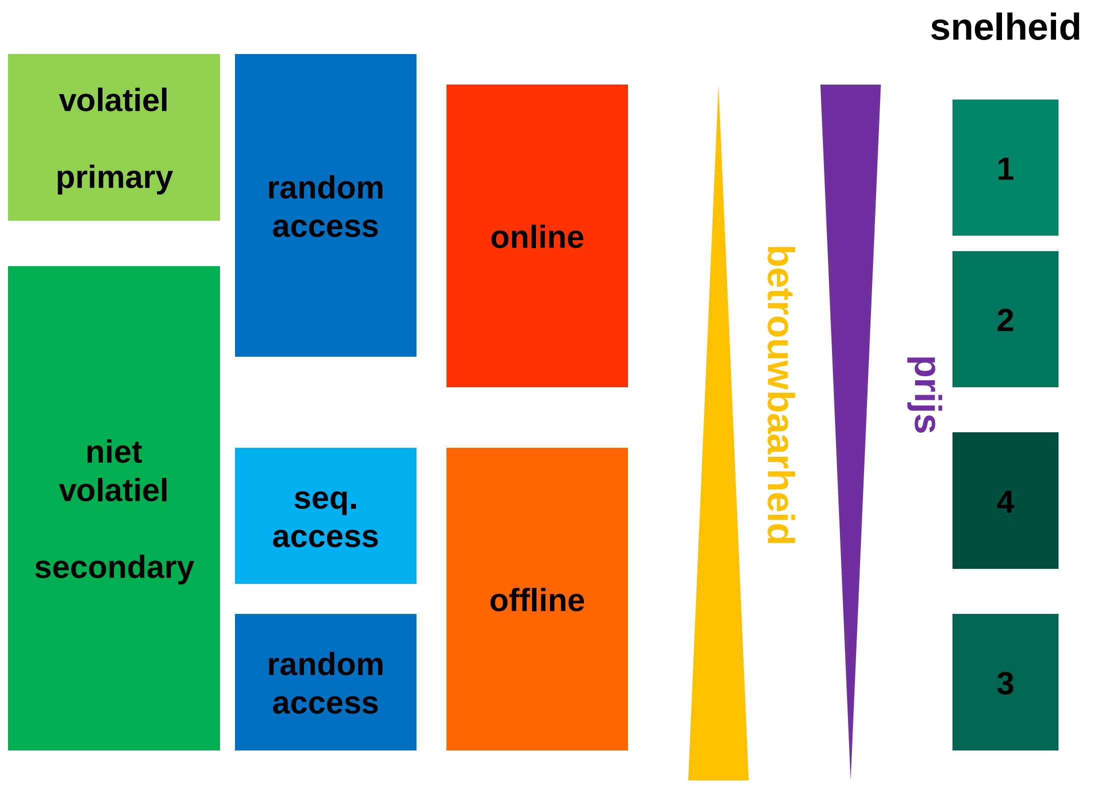
## Recovery

**recovery** is het proces waarbij
**een db wordt teruggebracht naar een correcte toestand**
wanneer er zich een failure voordoet

- waar zit de data?
- welke failures kunnen zich voordoen?

# Media

- waar zit data?
  - main memory
  - magnetic disk
  - tape
  - optical disk



**stabiele opslag**: replicatie op verschillende plaatsen met onafhankelijke failure modes

# Soorten failures

- **system crash**
  - hardware of software errors
  - verlies van gegevens in main memory
- **media failure**
  - vb. disk head crash
  - verlies van gegevens in secondary storage
- **software error in application**
  - vb. logische fout die transaction doet falen
- **natuurlijke 'rampen'**
  - vb. brand, aardbeving
- **slordigheid**
  - vb. onopzettelijk wissen van gegevens door gebruiker of db administrator
- **sabotage**
  - vb. opzettelijk wissen of corrupteren van gegevens of infrastructuur (sw/hw)

## Transactions and recovery

- eenheid voor recovery is een transaction
- recovery manager staat in voor
  - atomiciteit (ACID)
  - duurzaamheid (ACID)
- moeilijkheid
  - schrijven naar een db is niet atomair
    - transaction kan committen zonder dat alle effecten al (permanent) in de db geregistreerd zijn

# High level vs low level operaties

- Example

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, new_salary)
```

- zoek adres van disk block die het record met primary key x bevat
- transfer disk block in een buffer in main memory
- kopieer de waarde voor salary uit de buffer in de variabele *salary*

- zoek adres van disk block die het record met primary key x bevat
- transfer disk block in een buffer in main memory
- kopieer de waarde van de *salary* variabele in de buffer
- schrijf de buffer weg naar disk

## Undo and redo

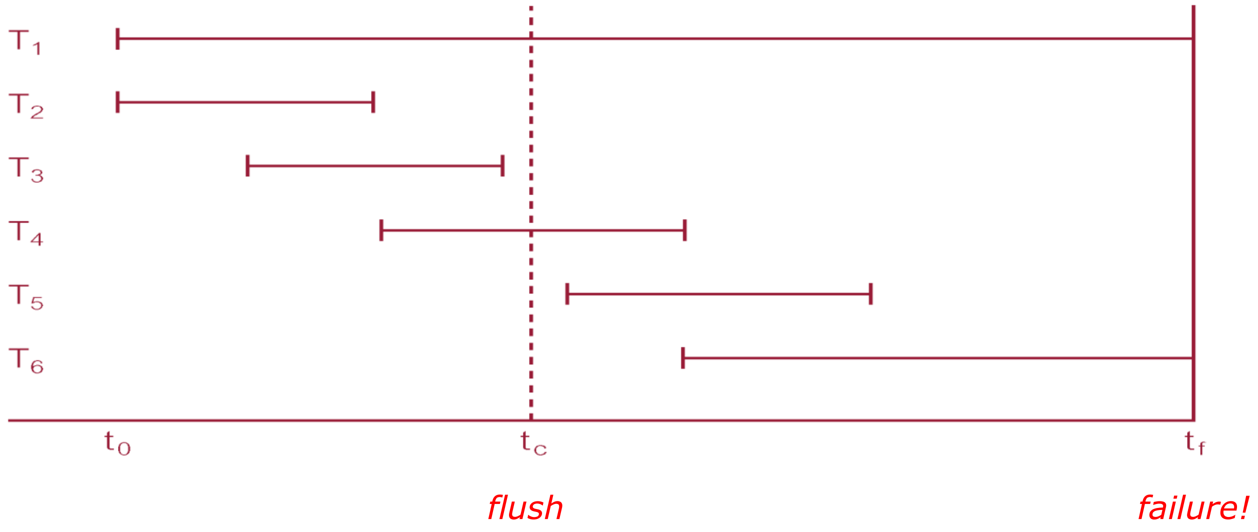
- enkel bij een 'flush' van de buffer is data permanent
  - flushing: data van primary storage overhevelen naar disk storage
- expliciete flush
  - force-writing
  - dit gebeurt via een commando, bv. bij commit
- impliciete flush
  - wanneer de buffers vol zijn

## Undo and redo

- wat bij failure tussen schrijven naar buffer and flushing?
  - transaction was reeds ge-commit
    - durability: **redo** de wijzigingen
    - aka **rollforward**
  - transaction was nog niet ge-commit
    - atomicity: **undo** de wijzigingen
      - partiële undo: undo van 1 transaction
      - globale undo: undo van alle actieve transactions
    - aka **rollback**

# Undo and redo

- Example



- bij failure
  - undo van T1 and T6
  - redo van T2, T3, T4, T5 (tijdstip flush niet bekend)

## Buffer management

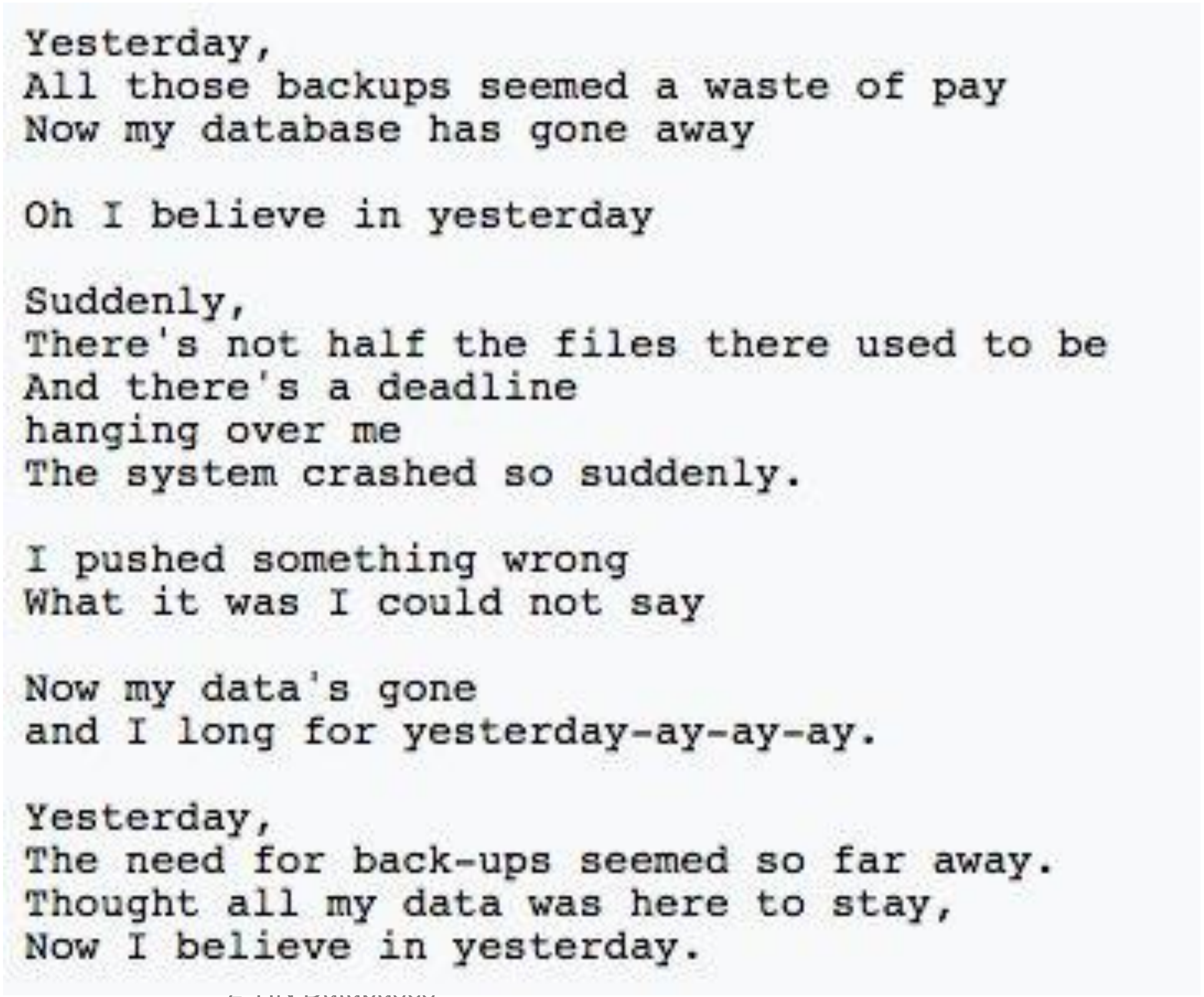
- buffer management omvat het beheer van transfer van buffers tussen main memory and disk
- praktisch:
  - inlezen tot buffer vol
  - replacement strategie voor force-write
    - FIFO first-in-first-out
    - LRU least recently used

- merk op: een pagina aanwezig in een buffer wordt nooit gelezen van disk

# Recovery faciliteiten

- het dbms biedt volgende diensten aan
  - back-up mechanisme
    - periodische back-ups van de db
  - logging mogelijkheden
    - op de hoogte blijven van de huidige toestand van transactions and db wijzigingen
  - checkpoint mogelijkheden
    - om lopende wijzigingen in de db permanent te maken
  - recovery manager
    - om de db in een consistente toestand te brengen na een failure

# Backup



Yesterday,
All those backups seemed a waste of pay
Now my database has gone away

Oh I believe in yesterday

Suddenly,
There's not half the files there used to be
And there's a deadline
hanging over me
The system crashed so suddenly.

I pushed something wrong
What it was I could not say

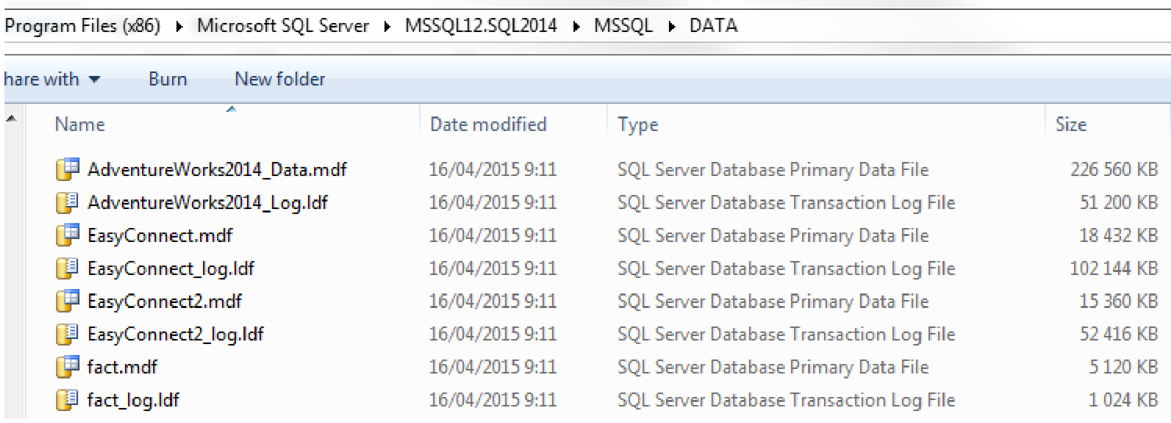
Now my data's gone
and I long for yesterday-ay-ay-ay.

Yesterday,
The need for back-ups seemed so far away.
Thought all my data was here to stay,
Now I believe in yesterday.

## Back-up mechanisme

- op regelmatige basis worden er automatisch reservekopieën van de db and de logfile aangemaakt
  - dit gebeurt zonder systeem te moeten stoppen
  - de kopieën worden bewaard op offline storage
- mogelijke benaderingen
  - complete back-up
  - incrementele back-up

# Logging



Program Files (x86) ▸ Microsoft SQL Server ▸ MSSQL12.SQL2014 ▸ MSSQL ▸ DATA

| Name | Date modified | Type | Size |
|---|---|---|---|
| AdventureWorks2014_Data.mdf | 16/04/2015 9:11 | SQL Server Database Primary Data File | 226 560 KB |
| AdventureWorks2014_Log.ldf | 16/04/2015 9:11 | SQL Server Database Transaction Log File | 51 200 KB |
| EasyConnect.mdf | 16/04/2015 9:11 | SQL Server Database Primary Data File | 18 432 KB |
| EasyConnect_log.ldf | 16/04/2015 9:11 | SQL Server Database Transaction Log File | 102 144 KB |
| EasyConnect2.mdf | 16/04/2015 9:11 | SQL Server Database Primary Data File | 15 360 KB |
| EasyConnect2_log.ldf | 16/04/2015 9:11 | SQL Server Database Transaction Log File | 52 416 KB |
| fact.mdf | 16/04/2015 9:11 | SQL Server Database Primary Data File | 5 120 KB |
| fact_log.ldf | 16/04/2015 9:11 | SQL Server Database Transaction Log File | 1 024 KB |

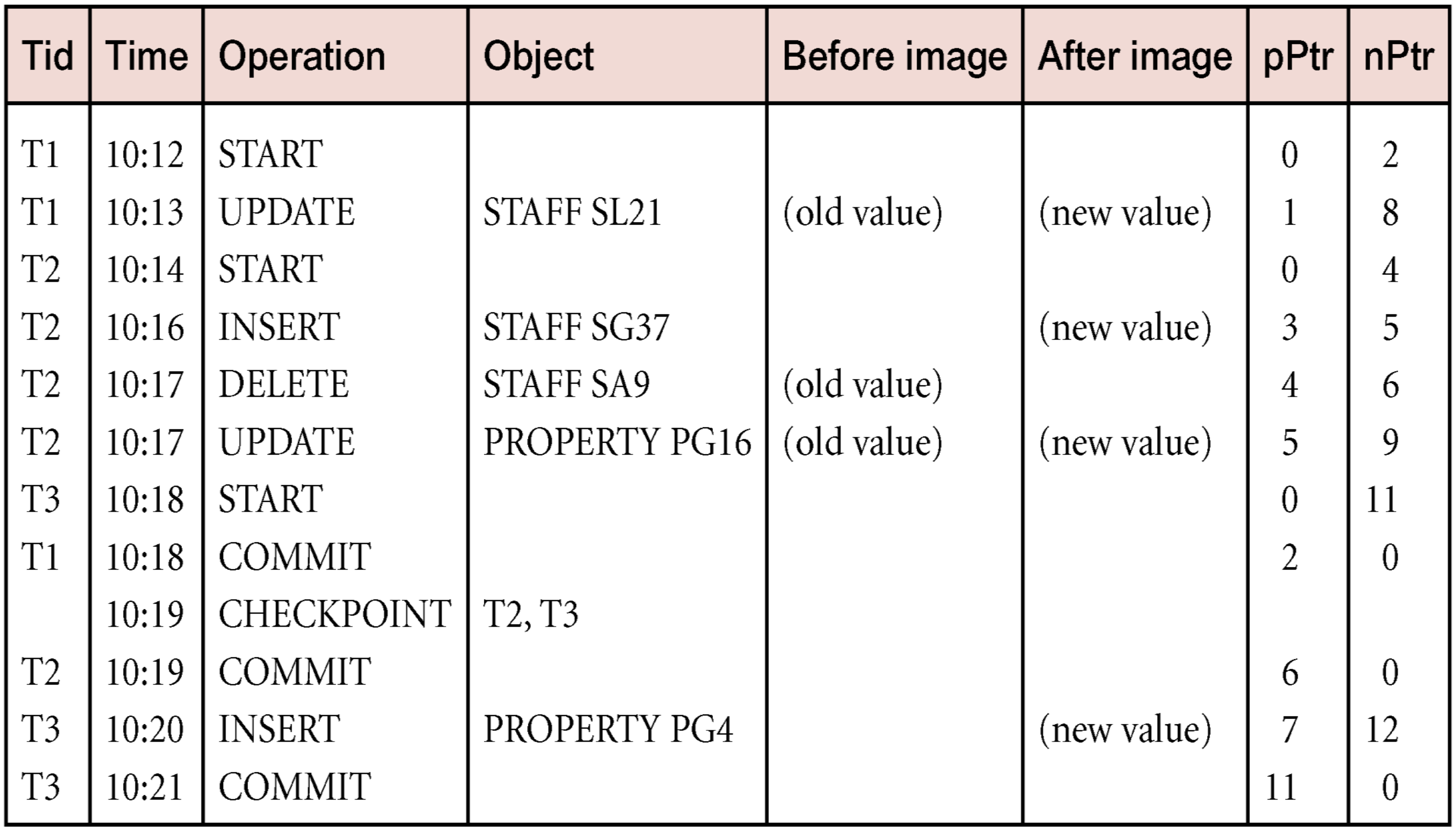
- **log** bevat mogelijks
  - **transaction records**
    - transaction id
    - type van log record *(transaction start, insert, update, delete, abort (rollback), commit)*
    - id van het gewijzigde data item *(enkel bij insert, update, and delete operaties)*
    - before-image
      - waarde data item vóór wijziging *(enkel bij update and delete operaties)*
    - after-image
      - waarde data item na wijziging *(enkel bij update and insert operaties)*
    - log management info
      - bv. pointers naar previous/next record van die transaction
  - **checkpoint records**

## Logging

- log wordt ook gebruikt voor andere doeleinden
  - performance monitoring, auditing
  - hiervoor is nog extra info in de log opgeslagen
- log is heel belangrijk
  - wordt in twee- of drievoud bijgehouden
  - ook offline
- log moet snel toegankelijk zijn
  - minor failures moeten direct opgelost worden
  - bevindt zich dus liefst ook op online storage
    - indien de grootte dit toestaat

## Logging

- Example log



| Tid | Time | Operation | Object | Before image | After image | pPtr | nPtr |
|---|---|---|---|---|---|---|---|
| T1 | 10:12 | START | | | | 0 | 2 |
| T1 | 10:13 | UPDATE | STAFF SL21 | (old value) | (new value) | 1 | 8 |
| T2 | 10:14 | START | | | | 0 | 4 |
| T2 | 10:16 | INSERT | STAFF SG37 | | (new value) | 3 | 5 |
| T2 | 10:17 | DELETE | STAFF SA9 | (old value) | | 4 | 6 |
| T2 | 10:17 | UPDATE | PROPERTY PG16 | (old value) | (new value) | 5 | 9 |
| T3 | 10:18 | START | | | | 0 | 11 |
| T1 | 10:18 | COMMIT | | | | 2 | 0 |
| | 10:19 | CHECKPOINT | T2, T3 | | | | |
| T2 | 10:19 | COMMIT | | | | 6 | 0 |
| T3 | 10:20 | INSERT | PROPERTY PG4 | | (new value) | 7 | 12 |
| T3 | 10:21 | COMMIT | | | | 11 | 0 |

# Checkpointing

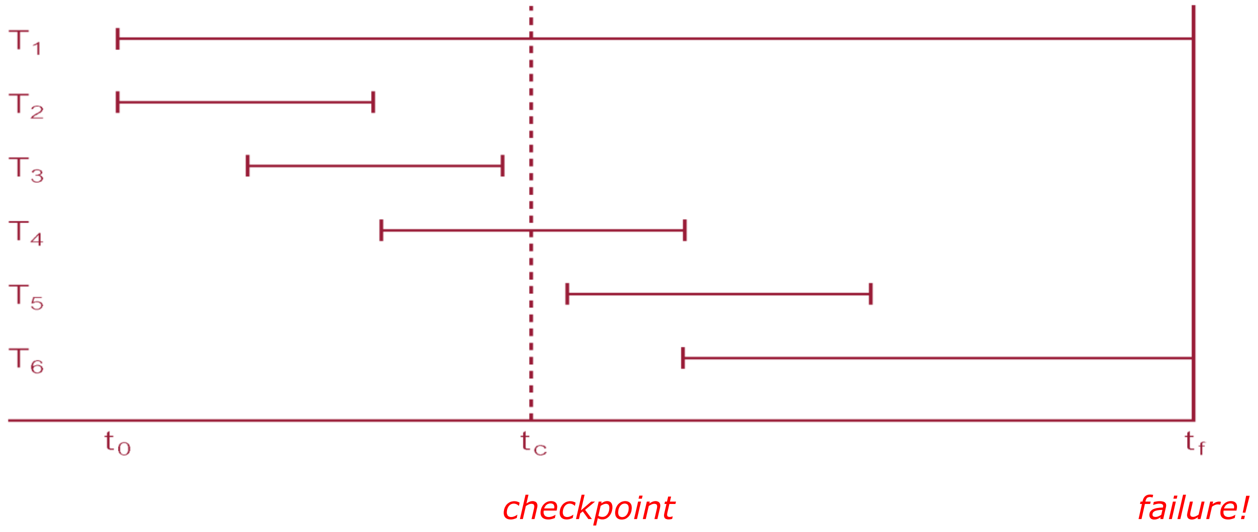
**een checkpoint** is een **synchronisatiepunt**
tussen de databank and de log,
op dit punt worden alle buffers ge-flushed

- checkpoints worden voorzien op vooraf ingestelde intervallen
  - bv. om de 15 minuten
- checkpointing omvat
  - alle log records in main memory wegschrijven naar disk
  - de gewijzigde delen van de buffers wegschrijven naar disk
  - een checkpoint record in de log registreren
    - dit record bevat id van alle transactions die actief zijn op het moment van checkpointing

Recovery - Recovery faciliteiten

## Checkpointing

- Example: recovery met checkpointing



- bij failure
  - T2 and T3: niets doen
  - T1 and T6: undo want waren actief op moment van crash
  - T4 and T5: redo want deze waren reeds ge-commit

## Recovery technieken

- soort recovery procedure die gevolgd wordt hangt af van de ernst van het probleem
  - serieuze (fysische) problemen
    - back-up restoren
    - wijzigingen van committed transactions (sedert de backup) die verloren gingen, herstellen adhv de log
  - problemen van inconsistentie
    - kan zonder back-up
    - undo/redo adhv de log's before and after images

## Deferred update

bij een **deferred recovery protocol** worden wijzigingen van een transaction niet weggeschreven naar de db zolang de transaction niet het commit-punt bereikt

- start van de transaction
  - registreer transaction start record in log
- bij een write actie
  - registreer dit in een log-record
    - enkel after-image, geen before image nodig
  - registreer dit niet in de db
- commit van de transaction
  - registreer transaction commit record in log
  - schrijf alle log records weg naar disk (flush de log **vóór** de volgende stap!) bij eerstvolgende checkpoint
  - commit, update de db adhv de log records
- abort van de transaction
  - negeer de log records, schrijf niets naar disk

# Immediate update

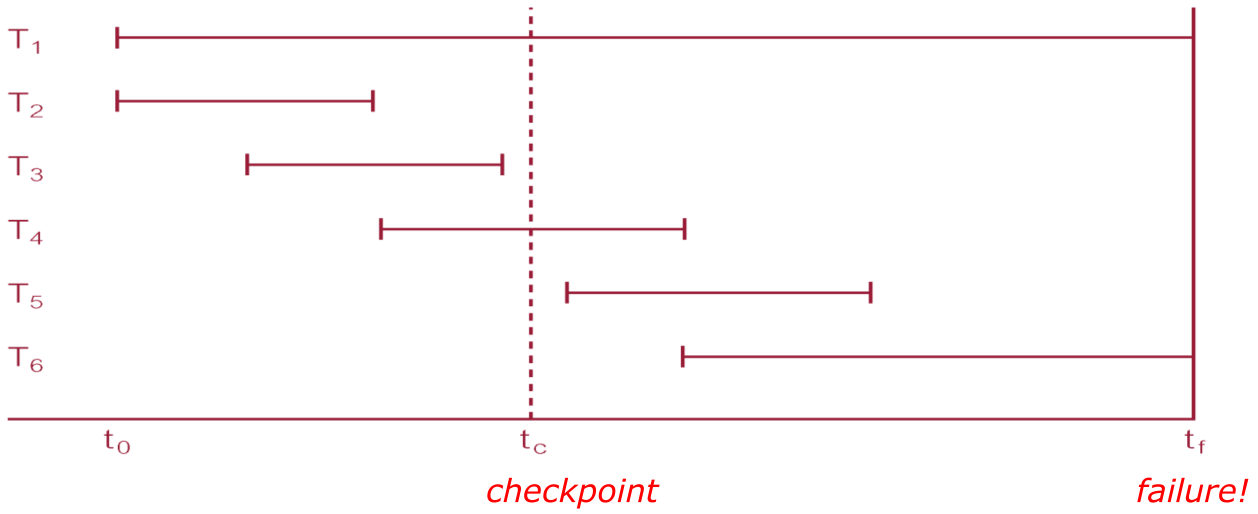
bij een **immediate update recovery protocol** worden wijzigingen van een transaction direct weggeschreven naar de db

- start van de transaction
  - registreer transaction start record in log
- bij een write actie
  - schrijf het **log-record naar disk**
    - het log-record bevat before-image, and after-image
  - bij flush van de buffers (bij eerstvolgende checkpoint) wordt de wijziging naar de db geschreven
- commit van de transaction
  - schrijf een transaction commit log record naar disk
- abort van de transaction
  - gebruik **before images** voor **undo**

write-ahead log strategie!

# Recovery technieken

- Example



| | deferred | immediate |
|---|---|---|
| T1 | niets | undo via before images |
| T2 | niets | niets |
| T3 | niets | niets |
| T4 | redo adhv after images | redo adhv after images |
| T5 | redo adhv after images | redo adhv after images |
| T6 | niets | undo via before images |

# Referentie handboek

GLOBAL EDITION
Database Systems
A Practical Approach to Design, Implementation, and Management
SIXTH EDITION
Thomas Connolly • Carolyn Begg
PEARSON

- Chapter 22 - 'Transaction Management'
- delen, paragrafen van dit hoofdstuk die niet werden behandeld in deze slides:
  - 22.2.2
    - View serializability
    - Testing for View serializability
  - 22.2.3
    - Concurrency recovery with index structures
    - Latches
  - 22.2.5
    - comparison of methods p597
  - 22.2.6 Multiversion timestamp ordering
  - 22.2.7 Optimistic techniques
  - 22.3.5 Recovery in a distributed DBMS
  - 22.4 Advanced transaction models
  - 22.5 Concurrency control and recovery in Oracle